**NAME: KEVIN ODHIAMBO**

**REGNO: BIT/2021/94874**

**UNIT CODE: BIT3209**

**UNIT TITLE: DESIGN AND ANALYSIS OF ALGORITHMS**

1. Counting Sort is efficient for sorting integers over a fixed range (like 0 to 9999) by counting occurrences and placing elements directly in the correct position.

2. Radix Sort works well on large ranges by sorting numbers digit by digit (from the least significant to the most significant digit) without needing comparison operations.

Python Code for Radix Sort:

from typing import List

def counting\_sort(arr: List[int], exp: int) -> List[int]:

n = len(arr)

output = [0] \* n

count = [0] \* 10

# Count occurrences of each digit

for i in range(n):

index = (arr[i] // exp) % 10

count[index] += 1

# Update the count array

for i in range(1, 10):

count[i] += count[i - 1]

# Build output array

for i in range(n - 1, -1, -1):

index = (arr[i] // exp) % 10

output[count[index] - 1] = arr[i]

count[index] -= 1

for i in range(n):

arr[i] = output[i]

return arr

def radix\_sort(arr: List[int]) -> List[int]:

max\_val = max(arr)

exp = 1

while max\_val // exp > 0:

counting\_sort(arr, exp)

exp \*= 10

return arr

B. Hospital Patient Queue Management

Concepts:

Queue (FIFO): Handles patient arrivals in order.

Stack (LIFO): Manages emergencies, with the latest emergency at the top.

Python Code for PatientQueue and EmergencyStack Classes:

from collections import deque

class PatientQueue:

def \_\_init\_\_(self):

self.queue = deque()

def enqueue(self, patient\_id):

self.queue.append(patient\_id)

def dequeue(self):

if self.queue:

return self.queue.popleft()

else:

return "Queue is empty"

class EmergencyStack:

def \_\_init\_\_(self):

self.stack = []

def push(self, emergency\_id):

self.stack.append(emergency\_id)

def pop(self):

if self.stack:

return self.stack.pop()

else:

return "No emergencies"

C. Storing User Profiles with Hashing

Concepts:

Hashing maps usernames to profiles. Collision strategies include:

Separate Chaining: Uses linked lists at each hash index.

Open Addressing: Finds alternative indexes within the array.

Python Code for UserHashTable:

class UserHashTable:

def \_\_init\_\_(self, size=10):

self.table = [[] for \_ in range(size)]

def hash\_function(self, username):

return hash(username) % len(self.table)

def add\_user(self, username: str, profile\_data: dict):

index = self.hash\_function(username)

for user in self.table[index]:

if user[0] == username:

user[1] = profile\_data

return

self.table[index].append((username, profile\_data))

def get\_user(self, username: str):

index = self.hash\_function(username)

for user in self.table[index]:

if user[0] == username:

return user[1]

return "User not found"

D. Library Catalog Management with Binary Search Trees (BST)

Concepts:

BSTs allow for efficient search, insertion, and deletion operations by keeping data sorted. Each left child is smaller, and each right child is larger than the parent node.

Python Code for BST Class:

class BSTNode:

def \_\_init\_\_(self, isbn, book\_data):

self.isbn = isbn

self.book\_data = book\_data

self.left = None

self.right = None

class BST:

def \_\_init\_\_(self):

self.root = None

def insert(self, isbn, book\_data):

self.root = self.\_insert\_recursive(self.root, isbn, book\_data)

def \_insert\_recursive(self, node, isbn, book\_data):

if not node:

return BSTNode(isbn, book\_data)

if isbn < node.isbn:

node.left = self.\_insert\_recursive(node.left, isbn, book\_data)

else:

node.right = self.\_insert\_recursive(node.right, isbn, book\_data)

return node

def search(self, isbn):

return self.\_search\_recursive(self.root, isbn)

def \_search\_recursive(self, node, isbn):

if not node or node.isbn == isbn:

return node

if isbn < node.isbn:

return self.\_search\_recursive(node.left, isbn)

return self.\_search\_recursive(node.right, isbn)

def delete(self, isbn):

self.root = self.\_delete\_recursive(self.root, isbn)

def \_delete\_recursive(self, node, isbn):

if not node:

return node

if isbn < node.isbn:

node.left = self.\_delete\_recursive(node.left, isbn)

elif isbn > node.isbn:

node.right = self.\_delete\_recursive(node.right, isbn)

else:

# Node with only one child or no child

if not node.left:

return node.right

elif not node.right:

return node.left

# Node with two children

temp = self.\_find\_min(node.right)

node.isbn = temp.isbn

node.book\_data = temp.book\_data

node.right = self.\_delete\_recursive(node.right, temp.isbn)

return node

def \_find\_min(self, node):

while node.left:

node = node.left

return node

E. File Indexing with B-Trees

Concepts:

B-Trees provide efficient indexing for large datasets by balancing the tree and minimizing the depth, making searches fast.

F. Route Optimization with Fibonacci Heaps

Concepts:

Fibonacci Heaps are ideal for priority queue operations, allowing quick decrease-key and extract-min operations, which makes them useful in Dijkstra’s algorithm for route optimization

Network Connectivity with Disjoint Set Unions (DSU)

Concepts:

Disjoint Set Union (DSU) is a data structure that helps manage and check connectivity between elements in different sets. DSU is useful in applications like network connectivity, where it can efficiently determine if two nodes belong to the same cluster.

Path Compression reduces the depth of trees by making nodes point directly to the root, which speeds up the find operation.

Union by Rank attaches the shorter tree under the root of the taller tree, keeping the overall tree as flat as possible for faster operations.

Python Code for DisjointSet Class:

class DisjointSet:

def \_\_init\_\_(self, n):

self.parent = list(range(n))

self.rank = [0] \* n

def find(self, u):

if self.parent[u] != u:

self.parent[u] = self.find(self.parent[u]) # Path compression

return self.parent[u]

def union(self, u, v):

root\_u = self.find(u)

root\_v = self.find(v)

if root\_u != root\_v:

# Union by rank

if self.rank[root\_u] > self.rank[root\_v]:

self.parent[root\_v] = root\_u

elif self.rank[root\_u] < self.rank[root\_v]:

self.parent[root\_u] = root\_v

else:

self.parent[root\_v] = root\_u

self.rank[root\_u] += 1

H. Algorithm Analysis for Optimal Search Times

Concepts:

Big-O (O) describes the upper bound or worst-case complexity of an algorithm, which helps understand the maximum time it could take.

Big-Omega (Ω) gives the lower bound, representing the best-case scenario.

Big-Theta (Θ) represents a tight bound, describing the average-case scenario, if both upper and lower bounds are the same.

Python Code for Analyzing Binary Search Performance:

import time

from typing import List

def binary\_search(arr: List[int], target: int) -> int:

left, right = 0, len(arr) - 1

while left <= right:

mid = (left + right) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

left = mid + 1

else:

right = mid - 1

return -1

def measure\_performance():

arr = list(range(100000)) # Large dataset

target = arr[len(arr) - 1] # Best case

start = time.time()

binary\_search(arr, target)

print("Best case:", time.time() - start, "seconds")

target = arr[len(arr) // 2] # Average case

start = time.time()

binary\_search(arr, target)

print("Average case:", time.time() - start, "seconds")

target = 1000000 # Worst case (not in array)

start = time.time()

binary\_search(arr, target)

print("Worst case:", time.time() - start, "seconds")

measure\_performance()

This function tests binary search across best, average, and worst cases to measure the time complexity. In the best case, the target is found in the first check; in the average case, it’s somewhere in the middle; and in the worst case, it’s absent, so the entire array is searched.

Additional Concepts

1. Big-O, Big-Theta, and Big-Omega Differences:

Big-O (O): Provides an upper limit, helpful for understanding the maximum growth rate of an algorithm.

Big-Omega (Ω): Gives a lower limit, which shows the minimum time required by the algorithm.

Big-Theta (Θ): A precise or tight bound, indicating an exact asymptotic behavior.

2. Big-O Notation Usage: Big-O is widely used because it highlights the worst-case performance, which is essential for system performance guarantees.

3. Stack vs. Queue Access Patterns:

Stack: LIFO (Last In, First Out) – ideal for tasks where the last added element needs to be accessed first.

Queue: FIFO (First In, First Out) – useful when elements need to be processed in the order they arrived.

.